8 浮動小数点数

- ♣ 2 進数の小数
- ♣ 浮動小数点表現 IEEE 754 (□ 型の正体)
- ♣ 浮動小数点数の演算

8.1 2 進数の小数点数

2 進数	···	100	10	1	0.1	0.01	0.001	0.0001	···
10 進数	···	(2^2) 4	(2^1) 2	(2^0) 1	(2^{-1}) □ 1/2	(2^{-2}) □ 1/4	(2^{-3}) □ 1/8	(2^{-4}) □ 1/16	···

例) b 11.11 = 2+1+ □ + □ = 3.75

例題 8.1 b0101.1011 を 10 進数に変換せよ.

整数部分は 5 である. 小数部分の変換方法は種々考えられる.

1) 定義通り計算する.

$$0.5 + 0.125 + 0.0625 = 0.6875$$

2) 定義に従って分数で計算する

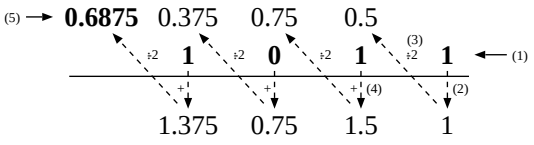
$$\frac{1}{2} + \frac{1}{8} + \frac{1}{16} = \frac{8+2+1}{16} = \frac{11}{16} = 0.6875$$

3) b1011 を 10 進数に直し, それを 4 回 (桁数の分だけ) 2 で割る (16 で割る)

$$\text{b1011} = 11$$

$$11 \xrightarrow{\div 2} 5.5 \xrightarrow{\div 2} 2.75 \xrightarrow{\div 2} 1.375 \xrightarrow{\div 2} 0.6875$$

4) 小数部分を計算する筆算も作れる



(1) 2 進数の小数部分
(2) 最初の桁はそのまま転記
(3) 2 で割る (4) 加算
<2 での除算と加算の繰り返し >
(5) 小数部分の 10 進表現

よって b0101.1011 = 5.6875

例題 8.2 6.4375 を 2 進数に変換せよ.

整数部分は b110. 小数部分の変換方法は種々考えられる.

1) 定義通り計算する

$$0.4375 = 0.25 + 0.1875 = 0.25 + 0.125 + 0.0625 = 2^{-2} + 2^{-3} + 2^{-4} = \text{b } 0.0111$$

2) 整数になるまで 2 倍を繰り返し, 得られた整数を 2 進数に変換する. 2 倍した回数だけ小数点をずらす.

$$0.4375 \xrightarrow{\times 2} 0.875 \xrightarrow{\times 2} 1.75 \xrightarrow{\times 2} 3.5 \xrightarrow{\times 2} 7$$

7 を 2 進数に変換すると b111

小数点を 4 つずらして b0.0111

※ ただし, 答えが循環小数になる場合 (後述) はこの方法ではツラい. 小数以下の桁数が m 桁に決まっている場合は, m 回 2 倍すればよい.

3) 「小数部を 2 倍する」を繰り返すと, 得られる整数部の系列が答えになっている.

$0.4375 \quad \times 2 = \quad 0.875 \quad \cdots 0$
$0.875 \quad \times 2 = \quad 1.75 \quad \cdots 1$
$0.75 \quad \times 2 = \quad 1.5 \quad \cdots 1$
$0.5 \quad \times 2 = \quad 1.0 \quad \cdots 1$
小数部が 0 になれば終了. 答えは 0.0111

10 進数の 0.1 はコンピュータ内部では「ちょうど」ではない

例えば, 10 進数の 0.1 を 2 進数に変換すると,

b0.0 0011 0011 0011 0011 $\cdots$

のように無限小数 ([ ] 小数) となる. 10 進数では有限の小数を, 2 進数では有限桁数で表現できるとは限らない. これは, 2 進数の基数 2 が 5 を約数に持たないためである. 例えば, 12 進数の 0.4 (つまり 1/3) を 10 進数で表そうとすると, 0.33333$\cdots$ のように循環小数になるが, これは 10 が 3 を約数に持たないためである.

無限小数を有限桁数の 2 進数で表現しようとすると, [ ] が生じる. 例えば 10 進数で 0.333333$\cdots$ を小数点以下 4 桁で表現しなければならないとすると, 0.3333 のように四捨五入や切り捨てをしなければならないが, この際に元の数との間に生じる誤差 (この場合は 0.00003333$\cdots$) が丸め誤差である.

このため, 例えば

```
double x = 0.1;
printf("x = %33.30f\n", x);      /* 全体 33 桁, 小数点以下 30 桁で表示 */
```

のような C プログラムを実行すると,

```
x = 0.100000000000000005551115123126
```

と表示される. 我々にはちょうどであっても, コンピュータ内部ではそうでないことは知っておこう.

8.2 指数表現

- 指数表現
  - 10 進数の指数表現 $315.834 = 3.15834 \times 10^2$

    有効数値を示す 3.15834 の部分 $\cdots$ [ ] 部 (mantissa)

    小数点の移動を表す 2 の部分 $\cdots$ [ ] 部 (exponent)
  - 2 進数の指数表現 $1010.1111 =$ [ ] $\times$ [ ]
- 正規化 (normalization)
  - 指数表現では, 同じ数を表す表現が複数存在

    $1100 \quad = \quad 110.0 \times 2^1 \quad = \quad 11.00 \times 2^2 \quad = \quad 1.100 \times 2^3 \quad = \quad 0.110 \times 2^4$

    (1) (2) (3) (4) (5)

    正規化 = 仮数部の整数部を [ ] にすること[1]

[1] 「整数部は 0 で小数第 1 位が 1 になっているもの」とする定義もある.

上の例で, 正規化されているのは □

☆ ただし □ は正規化できない

- 隠しビット (hidden bit)

  正規化された数の整数部は必ず □ → 内部表現ではこのビットを省略して 1 ビットを節約する
  このような表現を economized form, 省略されたビットを隠しビットという.

  例) 1.0110×2^5 を記憶する際の仮数部の表現は □

- バイアス表現 (エクセス表現)

  ☆ 正負の整数を表現する補数表現以外の方法の 1 つ (指数部の表現によく用いる)

  – n ビットのバイアス値 B のバイアス表現 (エクセス B 表現とも言う)

    $= n$ ビットの符号無し整数から B を引いた数を表現する

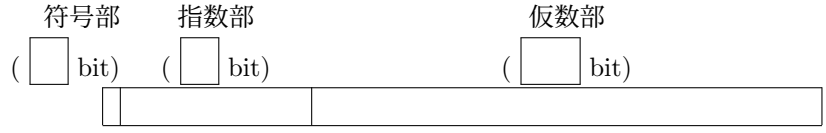
  例) バイアス値 $B = 6$ の 4 ビットバイアス表現

バイアス表現	表現する数		バイアス表現	表現する数	
0000	□	$(0-6)$	1000	□	$(8-6)$
0001	□	$(1-6)$	1001	□	$(9-6)$
0010	□	$(2-6)$	1010	□	$(10-6)$
0011	□	$(3-6)$	1011	□	$(11-6)$
0100	□	$(4-6)$	1100	□	$(12-6)$
0101	□	$(5-6)$	1101	□	$(13-6)$
0110	□	$(6-6)$	1110	□	$(14-6)$
0111	□	$(7-6)$	1111	□	$(15-6)$

8.3 IEEE 標準規格

浮動小数点数の標準規格 IEEE 754 [2]

- 32 ビット表現 (単精度 (single precision), C 言語の □ 型)



  – 符号部は, 正なら □, 負なら □

  – 指数部は, バイアス表現 (バイアス値 □)

    ∗ ただし, 0000 0000 と 1111 1111 は特殊用途に用いるので指数部の値は □ ∼ □

指数部	表現する数		備考
0000 0000	$0-127=$	(-127)	特殊用途用に使用
0000 0001	$1-127=$	-126	指数の最小
0000 0010	$2-127=$	-125	
···	···		
1111 1101	$253-127=$	126	
1111 1110	$254-127=$	127	指数の最大
1111 1111	$255-127=$	(128)	特殊用途に使用

[2] IEEE は電気・電子関係の国際的な学会で「アイ・トリプル・イー」と読む. 正式名称は The Institute of Electrical and Electronics Engineers, Inc.

– 仮数部は, [　　] ビットを使用

– 表現できる (正規化数) の範囲

絶対値最大 $\pm 1.11111111111111111111111 \times$ [　　] $= \pm 3.40282347 \times 10^{38}$

絶対値最小 $\pm 1.00000000000000000000000 \times$ [　　] $= \pm 1.17549435 \times 10^{-38}$

☆ C 言語で `FLT_MAX` が `3.40282347e+38`, `FLT_MIN` が `1.17549435e-38` となっているのはこのため

例題 8.3 単精度浮動小数点表現で xC1E0000 はどのような数を表すか.

xC1E00000

= b 1100 0001 1110 0000 0000 0000 0000 0000

	符号	指数部	仮数部
=	1	10000011	11000000000000000000000

– 符号ビットが 1 なので符号は [　　] である.

– 指数部は符号無しの数と見れば b 10000011 = 131 なので,

127 のバイアスを考慮すると [　] を表す.

– 仮数部は隠しビットを復元すると b 1.11 であり, b 1.11 = 1.75 を表す.

したがって, この数は [　　] (= [　　]) である.

- 正規化数以外の表現 (32bit)

1. ゼロ (zero)

ちょうど [　] を表す

符号	指数部	仮数部	
0	00000000	00000000000000000000000	… [　]
1	00000000	00000000000000000000000	… [　]

2. 無限大

符号	指数部	仮数部	
0	11111111	00000000000000000000000	… [　　]
1	11111111	00000000000000000000000	… [　　]

3. NaN ([　　] ; [　　])

[　　] の結果や [　　] 等を表す

符号	指数部	仮数部
0	11111111	0000000 ··· 0000000 以外
1	11111111	0000000 ··· 0000000 以外

4. 非正規化数 (denormalized numbers)

正規化数と [　　] の間の数をできる限り表現するのに用いる

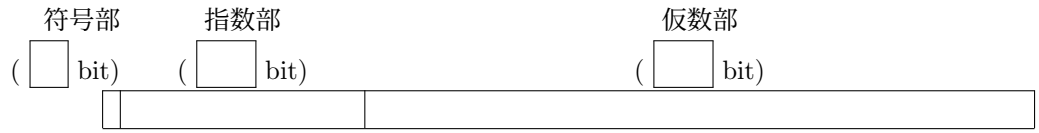
正規化数 (絶対値最小)　$1.00000000000000000000000 \times 2^{-126}$

非正規化数 (例)　$0.00010111001000110110010 \times 2^{-126}$

符号	指数部	仮数部
0	00000000	0000000 ··· 0000000 以外
1	00000000	0000000 ··· 0000000 以外

仮数部を m とすると, $0.m \times 2^{-126}$ を表す

- 64 ビット表現 (倍精度 (double precision)) , C 言語の [　　] 型)



  - 指数部のバイアスは 1023 で, 指数の範囲は $-1022 \sim 1023$
  - 表現できる (正規化数) の範囲

    絶対値最大 $\pm 1.11111111\cdots111 \times 2^{1023} = \pm 1.7976931348623157 \times 10^{308}$

    絶対値最小 $\pm 1.00000000\cdots000 \times 2^{-1022} = \pm 2.2250738585072014 \times 10^{-308}$

    ☆ C 言語では `DBL_MAX` が `1.7976931348623157e+308`, `DBL_MIN` が `2.2250738585072014e-308`

- 有効数値

  単精度: 2 進 23 桁 $\Rightarrow$ 10 進で約 [　] 桁　　$(23/10 * 3 = 6.9)$

  倍精度: 2 進 52 桁 $\Rightarrow$ 10 進で [　] 桁強　　$(52/10 * 3 = 15.6)$

  ☆ $2^{10} = 1,024 \fallingdotseq 10^3$ だから, 2 進数の 10 桁が 10 進数のほぼ [　] 桁になる. 従って, 2 進数の n 桁はおよそ 10 進数の [　　] 桁に相当する.

8.4 浮動小数点数の演算

8.4.1 計算結果の丸め

丸め ([　　]) … 計算結果を所望の桁数の数値にすること (例: [　　])

偶数丸め

- 単純な四捨五入

0.5 未満	0.5	0.5 超
切り捨て	**切り上げ**	切り上げ

0.5 を必ず切り上げとすると, 丸め後の数は平均的に大きくなる (特に桁数が少ない場合顕著)

.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	平均
−	捨	捨	捨	捨	**入**	入	入	入	入	
± 0.0	-0.1	-0.2	-0.3	-0.4	$\mathbf{+0.5}$	$+0.4$	$+0.3$	$+0.2$	$+0.1$	$\mathbf{+0.05}$

- 偶数丸め … 0.5 の切り上げ/切り捨てを半々にする

0.5 未満	0.5		0.5 超
	整数部が偶数	**整数部が奇数**	
切り捨て	**切り捨て** (結果は偶数)	**切り上げ** (結果は偶数)	切り上げ

例) $13.50 \rightarrow$ [　]
$14.50 \rightarrow$ [　]
$14.51 \rightarrow$ [　]　　$14.49 \rightarrow$ [　]

IEEE754 の丸め (4 つのモードが規定されている)

1. Unbiased (別名 [　　] 丸め) ← これがデフォルト
   - 最も近い値に丸める (要は原則として [　　])
   - 表現可能な 2 数のちょうど中間の数は LSB が [　　] になるように丸める

   例) 1010.011 → [　　]　　1010.110 → [　　]

   1100.100 → [　　]　　1101.100 → [　　]
2. Towards zero: (0 方向に丸める)
3. Towards negative infinity: ($-\infty$ 方向に丸める)
4. Towards positive infinity: ($+\infty$ 方向に丸める)

例)

元の数	2. (0 方向)	3. ($-\infty$ 方向)	4. ($+\infty$ 方向)
$+1010.011$	$+1010$	$+1010$	$+1011$
-1010.011	-1010	-1011	-1010

8.4.2 加減算

手順

1. 桁を [　　]

   2 数の指数部を比較し [　　] に合わせる

   その分仮数部を [　　]

   例)　1.111×2^5 ⇒ 1.111×2^5

   $+)$ 1.001×2^3　　$+)$ [　　] $\times 2^5$

2. 加算・減算を行う

   例)　1.111×2^5

   $+)$ 0.01001×2^5

   10.00101×2^5

3. 結果を [　　] する

   例) $10.00101 \times 2^5 \Rightarrow 1.000101 \times 2^6$

4. 結果を仮数のビット数に [　　]

   例) $1.000101 \times 2^6 \Rightarrow$ [　　]

   ※ 丸めの結果桁上がりすれば, 再度正規化が必要

8.4.3 乗算

手順

1. 仮数部は乗算を，指数部は [　　] を行う

   例) $(1.110 \times 2^5) \times (1.010 \times 2^3) = (1.110 \times 1.010) \times$ ([　　])

$$
\begin{array}{rcccccccc}
 & & & & 1 & 1 & 1 & 0 \\
\times) & & & & 1 & 0 & 1 & 0 \\
\hline
 & & & & 0 & 0 & 0 & 0 \\
 & & & 1 & 1 & 1 & 0 & \\
 & & 0 & 0 & 0 & 0 & & \\
 & 1 & 1 & 1 & 0 & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0
\end{array}
\quad \Rightarrow \quad \boxed{} \times \boxed{}
$$

2. 結果を正規化する

   例) $10.001100 \times 2^8 \quad \Rightarrow \quad 1.0001100 \times 2^9$

3. 結果を仮数のビット数に丸める

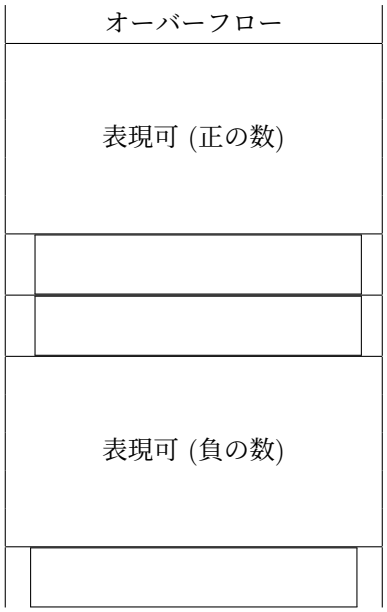
   例) $1.0001100 \times 2^9 \quad \Rightarrow$ $\boxed{}$

☆ 一般に, 浮動小数点数の演算は, 整数の演算よりも複雑で時間がかかる. 高性能のプロセッサでは浮動小数点専用の演算器を備えているが, そうでないプロセッサでは, なるべく浮動小数点演算を避けることが多い.

MFLOPS

Mega FLoating point Operations Per Second の略で, 1 秒間に浮動小数点演算を何 100 万回実行できるかを表す. スーパーコンピュータやグラフィックス用プロセッサ等, 浮動小数点演算を多用するコンピュータの性能指標に用いられる. コンピュータの性能向上に伴い, $\boxed{}$ FLOPS, $\boxed{}$ FLOPS, $\boxed{}$ FLOPS という単位も登場している.

8.4.4 オーバフローとアンダーフロー



← 絶対値最大の正の正規化数 (+FLT_MAX, +DBL_MAX)

← 絶対値最小の正の正規化数 (+FLT_MIN, +DBL_MIN)

← ± 0

← 絶対値最小の負の正規化数 (-FLT_MIN, -DBL_MIN)

← 絶対値最大の負の正規化数 (-FLT_MAX, -DBL_MAX)

アンダーフローが起こる例 (単精度浮動小数点数の場合, 絶対値最小の正の数は 1.0×2^{-126})

$1.0 \times 2^{-125} \times 1.0 \times 2^{-8} =$ $\boxed{}$ → アンダーフロー

$1.001 \times 2^{-126} - 1.000 \times 2^{-126} =$ $\boxed{}$ → アンダーフロー

オーバーフロー → 割り込みを起こすか, 正/負の [　　] にしてしまう

アンダーフロー → 割り込みを起こすか, ゼロか [　　] にしてしまう

☆ 非正規化数を用いれば，正規化数に比べて精度は劣るが計算を継続できる．これを段階的アンダーフロー (gradual underflow) と呼ぶ.

8.4.5 計算の誤差

1. 丸め誤差

   [　　] で計算を行うために生じる誤差

   例) [　　] の結果を有限桁で正確に表現することはできない

   例) [　　] を 2 進数では有限桁で正確に表現することはできない

2. 打ち切り誤差

   極限計算を [　　] で打ち切るために生じる誤差

   例) $\frac{1}{2^1}+\frac{1}{2^2}+\frac{1}{2^3}+\cdots$ を無限に計算することはできない.

3. 桁落ち

   値がほぼ等しい 2 数の [　　] を行うと [　　] が減少する

   例)
   $$\begin{array}{rll} & 1.101\times 2^5 & \text{(有効桁数 4 桁)} \\ -) & 1.100\times 2^5 & \text{(有効桁数 4 桁)} \\ \hline & \square\times 2^5 & \text{(有効桁数 } \square \text{ 桁)} \end{array}$$

4. 情報落ち

   絶対値の差が大きい 2 数の [　　] を行うと絶対値の小さい数が [　　] されてしまう

   例)
   $$\begin{array}{rll} & 1.111\times 2^{10} & \\ +) & 1.001\times 2^3 & \\ \hline & & \end{array} \quad \Rightarrow \quad \begin{array}{rl} & 1.111\times 2^{10} \\ +) & 0.0000000001001\times 2^{10} \\ \hline & \square\times 2^{10} \end{array}$$

☆ これらの特性を知って, 演算の順序を考慮する必要がある

例) 情報落ちを防ぐためには, 絶対値の [　　] から順に計算する

Nagisa ISHIURA